International Journal of Information Security manuscript No.
(will be inserted by the editor)

Shujing Wang · Yan Zhang

Handling Distributed Authorization with Delegation through Answer Set Programming

Abstract Distributed authorization is an essential issue in computer security. Recent research shows that trust management is a promising approach for the authorization in distributed environments. There are two key issues for a trust management system: how to design an expressive high-level policy language and how to solve the compliance-checking problem [5,6], where ordinary logic programming has been used to formalize various distributed authorization policies [19,20]. In this paper, we employ Answer Set Programming to deal with many complex issues associated with the distributed authorization along the trust management approach. In particular, we propose a formal authorization language $\mathcal{AL}$ providing its semantics through Answer Set Programming. Using language $\mathcal{AL}$, we can not only express nonmonotonic delegation policies which have not been considered in previous approaches, but also represent the delegation with depth, separation of duty, and positive and negative authorizations. We also investigate basic computational properties related to our approach. Through two case studies. we further illustrate the application of our approach in distributed environments.

Keywords Access control · trust management · authorization · delegation · answer set programming · knowledge representation · nonmonotonic reasoning

1 Introduction

Access control is an important topic in computer security research. It provides availability, integrity and confidentiality services for information systems. The traditional access control process includes identification, authentification and authorization. With the development of Internet, there are increasing applications that require distributed authorization decisions. For instance, in the application of electronic commerce, many organizations use the Internet (or large Intranets) to connect offices, branches, databases, and customers around the world and share their resources with other organizations. One essential problem among those distributed applications is about how to make authorization decisions, which is significantly different from that in centralized systems or even in distributed systems which are closed or relatively small. In traditional scenarios, the authorizer owns or controls the resources, and each entity in the system has a unique identity. Based on the identity and access control policies, the authorizer is able to make his/her authorization decision. In a distributed or multi-centralized authorization environment, however, there are more entities in the system, which can be both authorizers and requesters, and probably are unknown to each other. Quite often, there is no central authority that everyone trusts, as the authorizer does not know the requester directly, he/she has to use the information from the third parties who know the requester better. He/She trusts these third parties only for certain things to certain degrees. The trust and delegation issues make distributed authorization different from traditional access control scenarios.

In recent years, the trust management approach, which was initially proposed by Blaze *et al.* in [5], has received a great attention in information security community, e.g. [5–7,18,20]. Under this approach public keys are viewed as entities to be authorized and the authorization can be delegated to third parties by credentials or certificates. This approach frames the authorization decision as follows:

> "Does the set C of credentials prove that the request r complies with the local security policy P?"

From above we can see that there are at least two key issues for a trust management system:

1. Designing a high-level policy language to specify the security policy, credentials, and request. It is expected that the language has richer expressive power and is more human-understandable.

Shujing Wang · Yan Zhang
School of Computing and Mathematics, University of Western Sydney, Penrith South DC, NSW 1797, Australia
E-mail: {shwang, yan}@scm.uws.edu.au

2. Finding a feasible approach for the compliance checking.

Several trust-management systems such as PolicyMaker [5], Keynote [8], SPKI/SDSI [9–12,22], Delegation Logic [20], and RT framework [19] have been developed. PolicyMaker [5] was the first trust management system. Its access policies and credentials are called *assertions* which can be written in any programming language. It initiates the proof of compliance by creating a "blackboard" for inter-assertion communication, and a proof is achieved if the blackboard contains an acceptance record indicating that a policy assertion approves the request. Keynote [8] is the second generation of trust management systems and was designed according to the same principles as PolicyMaker. Instead of writing policy and credentials in a general-purpose procedural language, it adopts a specific expression. The both systems do not provide the negative authorization and re-delegation control. On the other hand, SPKI (Simple Public Key Infrastructure) [11] and SDSI (Simple Distributed Security Infrastructure) [22] were developed independently. Both of them were motivated by the inadequacy of public-key infrastructures based on global name hierarchies, such as X.509 [14] and Privacy Enhanced Mail (PEM) [16]. Later, SPKI and SDSI merged into a collaborative effort, SPKI/SDSI 2.0. SPKI/SDSI 2.0 has two kinds of certificates, name-definition certificates and authorization certificates. A name cert binds a local name to a principal or a more complex name. Name certs are used to resolve names to principals. An auth cert delegates a certain permission from a principal (the cert's issuer) to the cert's subject. SPKI/SDSI can deal with the *k-out-of-n* structures and handle certain types of nonmonotonic policies based on validity field of auth certificates. It controls whether the authorization should be delegated again or not, but there is no delegation depth control. Delegation Logic and RT framework, on other hand, both adopt logic programming based languages for representing security policies. D1LP [20], the monotonic version of Delegation Logic can express the delegation depth and complex principles including *k-out-of-n* structures using DATALOG as the sematic foundation. RT framework is a role-based trust management framework which includes languages RT_0, RT_1, RT_2, RT^D, and RT^T, where RT^D, and RT^T can be used together or separately, with RT_0, RT_1 or R_2. The semantic foundation of RT is DATALOG with constraints which enables RT to express the authorization regarding structured resources and separation of duty policies.

Although the existing trust management systems may express rich delegation and authorization policies, we observe that one important issue they do not consider is to express nonmonotonic policy and its related problems. For example, neither D1LP nor RT framework can deal with nonmonotonic reasoning. In the real world applications, many security policies have nonmonotonic features for decision making (e.g. authorization decision, delegation decisions including partial delegation), if there is no information to refute them.

Let us consider the following scenarios:

Scenario 1: In a large commercial organization, the system administrator trusts department managers and delegate them the privilege of accessing the file server with depth 1. Then managers give the privilege to the staff who are not in holiday in their departments and make them access the file server.
Scenario 2: In a hospital database, there is a table in which there is detailed information for its doctors, such as name, education background, specialized area, salaries and so on. The database administrator delegates patients to read all information about doctors except their salaries.
Scenario 3: A bank requires two cashiers to approve a transaction requested by customers if they do not have a bad credit history.

It is easy to see that all above scenarios have not only nonmontonic reasoning features, but also involve complex delegation and authorization controls. In Scenario 1, a department manager will make a positive authorization if there is no information stating that a staff in his/her department is on holidays. However the ordinary staff can not obtain delegation right to re-delegate this privilege because of the delegation depth control. Scenario 2 is a partial delegation decision which can be made if the requested resource from patients is not the doctor's salary. Scenario 3 is a separation of duty authorization in which the privilege to approve a transaction is delegated to two cashiers that should be specified using a dynamic threshold structure. The authorization decision will be made by two cashiers together if the customer requesting the transaction has not a bad credit history.

DAP (Delegable Authorization Program) proposed by Ruan et al. [25], is a logic program based formulation to support delegable authorizations. *DAP* permits negation as failure, classic negation, and rules inheritance and also provides a conflict resolution method for authorization conflicts. Although *DAP* provides nonmonotonic features in delegation reasoning, it does not have a flexible delegation control mechanism, which limits its expressive power to handle complex authorization and delegation representations. For example, Scenarios 1 and 2 involve the delegation depth control and partial delegation respectively, but they cannot be represented by *DAP*. On the other hand, as illustrated in Scenario 3, separation of duty plays an important role in authorization policy representation, and this feature cannot be expressed by *DAP* either. Finally, *DAP* has a difficulty to represent threshold structure in delegation specification as described in Scenario 3.

In this paper, we develop a formal language $\mathcal{AL}$ with nonmonotonic features, which is based on Answer Set

Programming, where negation as failure is used to implement nonmonotonic reasoning. $\mathcal{AL}$ also enables both positive and negative authorization which make policies more flexible. Most importantly, our proposed approach preserves all desirable features from existing trust management systems such as delegation with depth control, structured resources, separation of duty, etc. and overcome the major limitations of *DAP*. The reasons we choose Answer Set Programming as the foundation of language $\mathcal{AL}$ are as follows:

1. Answer Set Programming implements nonmonotonic reasoning through negation as failure. Nonmonotonic reasoning was developed to model commonsense reasoning used by human beings. A language with nonmonotonic features is much easier to specify security policies which is close to the natural language.
2. The highly efficient solvers for Answer Set Programming have been implemented, such as `Smodels, dlv` etc. This is an important reason that Answer Set Programming has been widely applied in product configuration, planning, constraint programming, cryptanalysis, etc. [2]. We need to indicate that `Smodels` supports some extended literals such as constraint literal and conditional literal which are particularly useful to express the static and dynamic threshold structures [26].

We should mention that although Answer Set Programming has been used in centralized authorization specifications [4,15], these previous work did not address the delegation aspect of distributed authorization.

The rest of this paper is organized as follows. Section 2 presents the syntax and expressive features of language $\mathcal{AL}$. Then Section 3 defines the semantics of language $\mathcal{AL}$ through Answer Set Programming. Section 4 provides scenarios to demonstrate the application of language $\mathcal{AL}$. Section 5 shows the computation properties of language $\mathcal{AL}$. Finally Section 6 presents the related work and concludes the paper.

2 An Authorization Language $\mathcal{AL}$

In this section, we first define the syntax of the authorization language $\mathcal{AL}$ and then illustrate its expressiveness via some examples.

2.1 Syntax of $\mathcal{AL}$

The authorization language $\mathcal{AL}$ consists of *entities, atoms, thresholds, statements, rules* and *queries*. The formal BNF syntax of $\mathcal{AL}$ is given in Figure 1. We explain the syntax in detail as follows.

Entities
In distributed systems, the entities include *subjects* who are authorizers owning or controlling resources and requesters making requests, *objects* which are resources and services provided by authorizers, and *privileges* which are actions executed on objects.

We define three types of constant entities, *subject, object* and *privilege*. Each constant entity is an element of three disjointed constant symbol sets, *SUB, OBJ*, and *PRIV*, where *SUB* is the set of subject constants, *OBJ* the set of object constants, and *PRIV* the set of privilege constants. The constant entity must start with a lower-case character.

Correspondingly each variable entity is an element of three disjointed variable symbol sets, V_{sub}, V_{obj}, and V_{priv} that range over the sets SUB, OBJ, and $PRIV$ respectively. The variable entities are prefixed with an upper-case character.

In the BNF of $\mathcal{AL}$, $\langle$*sub-con*$\rangle$, $\langle$*obj-con*$\rangle$, $\langle$*priv-con*$\rangle$, $\langle$*sub-var*$\rangle$, $\langle$*obj-var*$\rangle$, and $\langle$*priv-var*$\rangle$ represent elements of the sets SUB, OBJ, $PRIV$, V_{sub}, V_{obj}, and V_{priv} respectively.

In language $\mathcal{AL}$, we provide a special subject, *local*. It is the local authorizer which makes the authorization decision based on local policy and credentials from trusted subjects.

Atoms
An atom is a function symbol with n arguments - generally 1, 2, or 3 constant or variable entities, to express a logical relationship between them. There are three types of atoms:

1. $\langle$*relation-atom*$\rangle$. An atom in this type is a 2-ary function symbol and expresses the relationship of two entities. We provide three relation atoms, *neq, eq,* and *below.* The atoms *neq* and *eq* denote two same type entities equal or not equal, and *below* to denote the hierarchy structure for objects and privileges. In most realistic systems, the data information is organized using hierarchy structure, such as file systems and object oriented database system. For example, $below(ftp, pub\text{-}services)$ denotes that ftp is one of $pub\text{-}services$.
2. $\langle$*assert-atom*$\rangle$. This type of atoms, denoted by *exp* $(a_1, \ldots, a_n)$, is an application dependant function symbol with n arguments, where n is 1, 2 or 3, to express one, two or three constant or variable entities and states the property of the subjects, or the relationship between entities. It is a kind of flexible atoms in language $\mathcal{AL}$. For example, $isaTutor(alice)$ denotes that *alice* is a tutor.
3. $\langle$*auth-atom*$\rangle$. The *auth-atom* is of the form,
   $right(\langle sign \rangle, \langle priv \rangle, \langle obj \rangle)$,
   in which *sign* is +, -, or $\Box$. It states $positive(+)$ privilege, $negative$(-) privilege, or both($\Box$) of them. When an auth atom is used in delegation statement, the *sign* is $\Box$ to denote both positive and negative authorizations. For example, $right(+, update, students)$

⟨*rule*⟩	::=	⟨*head-stmt*⟩ [*if* [⟨*list-of-body-stmt*⟩]	
		[*with absence* ⟨*list-of-body-stmt*⟩]]	(1)
⟨*head-stmt*⟩	::=	⟨*relation-stmt*⟩ | ⟨*assert-stmt*⟩ |	
		⟨*auth-stmt-head*⟩ | ⟨*delegate-stmt-head*⟩	(2)
⟨*list-of-body-stmt*⟩	::=	⟨*body-stmt*⟩ | ⟨*body-stmt*⟩, ⟨*list-of-body-stmt*⟩	(3)
⟨*body-stmt*⟩	::=	⟨*relation-stmt*⟩ | ⟨*assert-stmt*⟩ |	
		⟨*auth-stmt-body*⟩ | ⟨*delegate-stmt-body*⟩	(4)
⟨*relation-stmt*⟩	::=	"*local*" *says* ⟨*relation-atom*⟩	(5)
⟨*assert-stmt*⟩	::=	⟨*sub*⟩ *asserts* ⟨*assert-atom*⟩	(6)
⟨*auth-stmt-body*⟩	::=	⟨*sub*⟩ *grants* ⟨*auth-atom*⟩ *to* ⟨*sub*⟩	(7)
⟨*auth-stmt-head*⟩	::=	⟨*sub*⟩ *grants* ⟨*auth-atom*⟩ *to* ⟨*sub-ext-struct*⟩	(8)
⟨*delegate-stmt-body*⟩	::=	⟨*sub*⟩ *delegates* ⟨*auth-atom*⟩ *with depth* ⟨*k*⟩ *to* ⟨*sub*⟩	(9)
⟨*delegate-stmt-head*⟩	::=	⟨*sub*⟩ *delegates* ⟨*auth-atom*⟩ *with depth* ⟨*k*⟩ *to* ⟨*sub-struct*⟩	(10)
⟨*relation-atom*⟩	::=	*below*(⟨*obj*⟩, ⟨*obj*⟩) | *below*(⟨*priv*⟩, ⟨*priv*⟩ |	
		neq(⟨*entity*⟩, ⟨*entity*⟩) | *eq*(⟨*entity*⟩, ⟨*entity*⟩)	(11)
⟨*assert-atom*⟩	::=	*exp*(⟨*entity-set*⟩)	(12)
⟨*auth-atom*⟩	::=	*right*(⟨*sign*⟩, ⟨*priv*⟩, ⟨*obj*⟩)	(13)
⟨*obj*⟩	::=	⟨*obj-con*⟩ | ⟨*obj-var*⟩	(14)
⟨*priv*⟩	::=	⟨*priv-con*⟩ | ⟨*priv-var*⟩	(15)
⟨*sub*⟩	::=	⟨*sub-con*⟩ | ⟨*sub-var*⟩	(16)
⟨*sub-set*⟩	::=	⟨*sub-con*⟩ | ⟨*sub-con*⟩, ⟨*sub-set*⟩	(17)
⟨*sub-struct*⟩	::=	⟨*sub*⟩ | "["⟨*sub-set*⟩"]" | ⟨*threshold*⟩	(18)
⟨*sub-ext-set*⟩	::=	⟨*dth*⟩ | ⟨*dth*⟩, ⟨*sub-ext-set*⟩	(19)
⟨*sub-ext-struct*⟩	::=	⟨*sub*⟩ | "["⟨*sub-set*⟩"]" | ⟨*threshold*⟩ | "["⟨*sub-ext-set*⟩"]"	(20)
⟨*entity*⟩	::=	⟨*sub*⟩ | ⟨*obj*⟩ | ⟨*priv*⟩	(21)
⟨*entity-set*⟩	::=	⟨*entity*⟩ | ⟨*entity*⟩, ⟨*entity-set*⟩	(22)
⟨*sign*⟩	::=	+ | − | □	(23)
⟨*k*⟩	::=	⟨*natural-number*⟩	(24)
⟨*threshold*⟩	::=	⟨*sth*⟩ | ⟨*dth*⟩	(25)
⟨*sth*⟩	::=	*sthd*(⟨*k*⟩, "["⟨*sub-set*⟩"]")	(26)
⟨*dth*⟩	::=	*dthd*(⟨*k*⟩, ⟨*sub-var*⟩, ⟨*assert-stmt*⟩)	(27)
⟨*query*⟩	::=	⟨*sub*⟩ *requests* (+, ⟨*priv*⟩, ⟨*obj*⟩) |	
		"["⟨*sub-set*⟩"]" *requests* (+, ⟨*priv*⟩, ⟨*obj*⟩)	(28)

Fig. 1 BNF for the Authorization Language $\mathcal{AL}$

indicates the positive *update* privilege on *students* table.

Statements

There are four types of statements, *relation statement, assert statement, auth statement*, and *delegation statement.* Only the local authorizer can issue the *relation statement* to denote the structured resources and privileges. We provide the body and head forms for auth statements and delegation statements.

Threshold

There are two types of threshold structures, static threshold and dynamic threshold.

The static threshold structure is of the form,

$$sthd(k, [s_1, s_2, \ldots, s_n]),$$

where k is the threshold value, $[s_1, s_2, \ldots, s_n]$ is the static threshold pool, and we require $k \leq n$ and $s_i \neq s_j$ *for* $1 \leq i \neq j \leq n$. This structure states that we choose k subjects from the threshold pool.

The dynamic threshold structure is of the form:

$dthd\ (k, S, \langle sub\rangle\ assert\ exp(\ldots, S, \ldots))$,

where S is a subject variable and we require that S is an argument of assert atom *exp*. This structure denotes we choose k subjects who satisfy the assert statement.

Rules
The rule is of the form,

$\langle head\text{-}stmt\rangle\ \ if\ \langle list\text{-}of\text{-}body\text{-}stmt\rangle,$
$with\ absence\ \langle list\text{-}of\text{-}body\text{-}stmt\rangle.$

The basic unit of a rule is a statement. Let h_0 be a head statement and b_i a body statement, then a rule is expressed as follows,

$h_0,\ if\ b_1,\ b_2, \ldots, b_m,\ with\ absence\ b_{m+1}, \ldots, b_n.$

In language $\mathcal{AL}$, a rule is a local authorization policy or a credential from other subjects and the issuer of the rule is the issuer of the head statement h_0. That is the reason why we limit the issuer structure in statements.

Query
Language $\mathcal{AL}$ supports single subject query and group subject query. They are of the forms:

sub requests $right(+, p, o)$, and
$[s_1, s_2, \ldots, s_n]$ requests $right(+, p, o)$.

Through group subject query, we implement *separation of duty* which is an important security concept. It ensures that a critical task cannot be carried out by one subject. If we grant an authorization to a group subject, we permit it only when all of subjects in the group request the authorization at the same time.

2.2 Characteristics of $\mathcal{AL}$

In this subsection, we present some examples to show the expressive power of $\mathcal{AL}$.

Structured resources
In the file system of a server in a university, there is a directory *postgraduate* which has one subdirectory for each postgraduate student, such as *alice*, *bob*, and so on.

local says $below(alice,\ postgraduate)$.
local says $below(bob,\ postgraduate)$.

Structured privileges
In a database system, there are a group of privileges *allrights* including *insert*, *delete*, and *select*.

local says $below(insert,\ allrights)$.
local says $below(delete,\ allrights)$.
local says $below(select,\ allrights)$.

Partial delegation and authorization
A firewall system protects the *allServices*, including *ssh*, *ftp*, and *http*. The administrator permits *ipA* to access all the services except *ssh* and delegates this right to *ipB* with two steps.

local delegates $right(\Box,\ access,\ X)$ with depth 2 to *ipB* if
local says $below(X,\ allServices)$, *local* says $neq(X,\ ssh)$.
local grants $right(+,\ access,\ X)$ to *ipA* if
local says $below(X,\ allServices)$, *local* says $neq(X,\ ssh)$.

Separation of duty
A company chooses to have multiparty control for emergency key recovery. If a key needs to be recovered, three persons are required to present their individual PINs. They are from different departments, *managerA*, a member of management, *auditorB*, an individual from auditing department, and *techC*, one individual from IT department.

local grants $right(+,\ recovery,\ k)$ to
$[\ managerA,\ auditorB,\ techC\ ]$.

Negative authorization
In a firewall system, the administrator *sa* does not permit *ipB* to access the *ftp* services.

sa grants $right(-,\ access,\ ftp)$ to *ipB*.

Nonmonotonic reasoning
In a firewall system, the administrator *sa* permit a *person* to access the *mysql* service if the human resource manager *hrM* asserts the person is a *staff* and not on holiday.

sa grants $right(+,\ access,\ mysql)$ to X if
hrM asserts isStaff(X),
with absence *hrM asserts onHoliday*(X).

3 Semantics of $\mathcal{AL}$

In this section, we first introduce Answer Set Programming (ASP) which is the foundation of language $\mathcal{AL}$, and an ASP based language $\mathcal{L}_{Ans}$. We define the semantics for language $\mathcal{AL}$ by translating it to language $\mathcal{L}_{Ans}$. We present function $TransRules(\mathcal{D}_{\mathcal{AL}})$ to translate an domain description $\mathcal{D}_{\mathcal{AL}}$ under $\mathcal{AL}$ into program $\mathcal{P}$ of $\mathcal{L}_{Ans}$, and function $TransRules(\mathcal{D}_{\mathcal{AL}})$ to translate query $\mathcal{Q}_{\mathcal{AL}}$ into program Π and ground literals $\varphi(+)$ and $\varphi(-)$. We use $\varphi(+)$ to denote positive right and $\varphi(-)$ to denote negative right. We solve a query based on $\mathcal{P}$, Π and φ via *Smodels*.

Definition 1 A *domain description* $\mathcal{D}_{\mathcal{AL}}$ of language $\mathcal{AL}$ is a finite set of rules.

Definition 2 The *size* of a *domain description* $\mathcal{D}_{\mathcal{AL}}$, $|\mathcal{D}_{\mathcal{AL}}|$ is the number of rules in it.

Definition 3 Given a domain description $\mathcal{D}_{\mathcal{AL}}$ and a query $\mathcal{Q}_{\mathcal{AL}}$ of language $\mathcal{AL}$, we define functions $TransRules(\mathcal{D}_{\mathcal{AL}}) = \mathcal{P}$ and $TransQuery(\mathcal{Q}_{\mathcal{AL}}) = <\Pi, \varphi(+), \varphi(-)>$. We say that query $\mathcal{Q}_{\mathcal{AL}}$ is *permitted, denied,* or *unknown* by the domain description $\mathcal{D}_{\mathcal{AL}}$ iff $(\mathcal{P} \cup \Pi) \models \varphi(+)$, $(\mathcal{P} \cup \Pi) \models \varphi(-)$, or $(\mathcal{P} \cup \Pi) \not\models \varphi(+)$ and $(\mathcal{P} \cup \Pi) \not\models \varphi(-)$ respectively.[1]

[1] Functions *TransRules* and *TransQuery* will be specified in section 3.3.

3.1 Answer Set Programming: An Overview

Now we give a brief overview of the answer set (stable model) semantics [3] and necessary terminologies in *Smodels* language [26], which includes the basic terms in logic programs and the extended stuff that consists of the special features of *Smodels*.

There are four different types of terms: *constants, variables, functions*, and *ranges* in *Smodels*. A *constant* is either a symbolic constant or numeric constant starting with a lower case letter. A *variable* is a string of letters and numbers starting with an upper case letter. A *function* is either a function symbol followed by a parenthesized argument list or a built-in arithmetical expression. A *range* is of the form:

start..end

where *start* and *end* are constant valued arithmetic expressions. A *range* is a notational shortcut that is mainly used to define numerical domains in a compact way.

An *atom* is of the form $p\,(a_1, \ldots, a_n)$ where p is a *n-ary* predicate symbol and $a_1, \ldots, a_n$ $(n \geq 0)$ are terms. Generally, a *literal* is either an *atom a* or its negation *not a*. We call it a *basic literal.* In *Smodels*, there are three extended literals, *constraint literals, weight literals*, and *conditional literals.* We do not consider the weight situation in language $\mathcal{AL}$. Then the *weight literal* is not discussed here.

A *constraint literal* is of the form:

$lower\ \{\ l_1,\ l_2, \ldots, l_n\ \}\ upper$

where *lower* and *upper* are arithmetic expressions and $l_1, \ldots, l_n$ are *basic* or *conditional literals.* A *constraint literal* is satisfied if the number of satisfied literals in the body of the constraint is between *lower* and *upper* (inclusive).

A *conditional literal* is of the form:

$p\,(X)\ :\ q\,(X)$

where $p\,(X)$ is any *basic literal* and $q(X)$ is a *domain predicate*. Formally, a predicate p of program Π is a *domain predicate* iff in the predicate dependency graph of Π every path starting from p is free of cycles that pass through a negative edge.

A rule is of the form:

$h \leftarrow l_1, \ldots, l_n.$

Where the literal h is the rule *head* and literals $l_1, \ldots, l_n$ form the rule *body.* In *Smodels*, h can be basic atom, positive conditional or constraint literal and $l_1, \ldots, l_n$ can be basic literal, conditional literal or constraint literal. *Smodels* improves its expressive power using rich literal forms. If the rule body is empty $(n = 0)$, the rule is called a *fact.* A rule is called a *Horn rule* if it does not have any negative literals. A normal *logic program* is a set of rules.

In [3], a logic program just including *Horn rules* is called $AnsProlog^{-not}$. A *normal logic program* is called $AnsProlog$ where negation as failure is allowed to occur in the rule's body. $AnsProlog$ programs are a superclass of $AnsProlog^{-not}$ programs in that they allow the operator *not*, *negation-as-failure* in the body of rules.

$AnsProlog^{-not}$ programs form the simplest class of declarative logic programs, and its answer set semantics usually can be defined using two ways, *model theoretical approach* and *iterated fixpoint approach.* The answer set semantics of $AnsProlog$ programs can be defined using neither the approach of minimal models nor that of iterated fixpoints. The problem is that in minimal model approach, $AnsProlog$ programs may have multiple minimal models and in the iterated fixpoint approach, an $AnsProlog$ program may not lead to a fixpoint. The answer set semantics of $AnsProlog$ programs is defined based on the Gelfond-Lifschitz transformation (also referred to as a reduct), as it was originally defined by Gelfond and Lifschitz in [13]. The detailed definition for them is referred to [3].

A program may have one, more than one, or no answer sets at all. For a given program Π and a ground atom φ, we say Π *entails* φ, denoted by $\Pi \models \varphi$, iff φ is in every answer set of Π.

3.2 The language $\mathcal{L}_{Ans}$

$\mathcal{L}_{Ans}$ is an ASP based language with answer set semantics. A program of $\mathcal{L}_{Ans}$ can be computed by *Smodels.* In this subsection, we first present the alphabet for language $\mathcal{L}_{Ans}$, and then give the propagation rules, authorization rules, and conflict resolution and decision rules in $\mathcal{L}_{Ans}$.

3.2.1 The language alphabet of $\mathcal{L}_{Ans}$

1. Entity Sort:
   There are three types of constant entities, *subject*, *object*, and *privilege.* The subject entity sort includes group subject entities introduced in the translation to denote a set of subjects. All the constant entities start with a lowercase characters.
   Accordingly, there are three disjointed variable sets, the sets of subject variables, object variables, and privilege variables that range over the constant entities respectively. The variable entities begin with a uppercase characters.
2. Function symbols:
   right(*sign*, *priv*, *obj*), where *sign* is $+$, $-$, or $\Box$, *priv* privilege sort, *obj* object sort.
   exp($a_1, \ldots,\ a_n$), where a_i is an entity sort, and *exp* is an application dependant assertion atom name. For example, $isDoctorOf(alice, tom)$, which denotes that *alice* is the doctor of *tom.*
   In *Smodels*, the above both functions are symbolic functions which just defines a new constant as an argument for the predicates in the application. We define them just to combine the related arguments

together to express a right or an assertion which are parameters for predicates *auth*, *delegate*, and *assert*. After the rules in the program are grounded, there are no any variables in both functions and they are just ordinary constant arguments for the related predicates.
max$(t_1, \ldots, t_n)$, where t_i's are integers. The function returns a biggest integer among t_i's.
min$(t_1, \ldots, t_n)$, where t_i's are integers. The function returns a smallest integer among t_i's[2].

3. Predicate symbols:
below$(arg1,\ arg2)$, where $arg1$ and $arg2$ are of the same *entity sort* to denote partial order relationship in a hierarchy structure. For example, below(read, write) means the privilege *read* is dominated by *write*.
assert$(issuer,\ exp(a_1, \ldots, a_n))$, where *issuer* is of *subject sort*, *exp* is an application dependant function of n arguments that are of *entity sort*.
auth$(issuer,\ grantee,\ right(sign, priv, obj), step)$, where *issuer* and *grantee* are both of *subject entity* sort, *step* is a natural number or variable which means how many steps the *right* goes through from *issuer* to *grantee*.
delegate$(issuer,\ delegatee,$ $right(sign, priv, obj),\ depth,\ step)$, where *issuer* and *delegatee* are of *subject entity* sorts, *depth*, and *step* are natural numbers or variables. *depth* states how far the *right* can be delegated further. *step* states how many steps the delegation has gone through.
req$(sub,\ right(+, priv, obj))$, where *sub* is of *subject entity* sort. It states that the *sub* requests the $right(+, priv, obj)$.
grant$(sub,\ right(sign, priv, obj))$, where *sub* is of *subject entity* sort. It states that the $right(sign, priv, obj)$ is granted to *sub*.
For the group subject query, we present predicate *ggrant* and *match*.
ggrant$(sub,\ right(sign, priv, obj))$, where *sub* is one of *subject group entities* introduced during the translation process. It states that the $right(sign, priv, obj)$ is granted to a set of subjects.
match$(sub, right(sign, priv, obj))$, where *sub* is one of *subject group entities* introduced during the translation process. It states that people requesting the *right* are exactly those who are authorized.
We also introduce some predicates for the authorization and conflict resolving rules.
exist_pos$(sub, right(+, priv, obj))$, where *sub* is of subject entity sort. It states there is positive *privilege* on *obj* for *sub*.
pos_far$(sub, right(+, priv, obj), step)$, where *sub* is of subject entity sort. It states that there is at least one negative authorization $right(-, priv, obj)$ for *sub* which has less steps than the positive authorization $right(+, priv, obj)$ with *step* for *sub*. For example, if both of $auth(s1, s2, right(+, read, file1), 4)$ and $auth(s3, s2, right(-,\ read, file1), 3)$ exist, we can get pos_far$(s2, right(+, read, file1), 4)$.
exist_neg$(sub, right(-, priv, obj))$, where *sub* is of subject entity sort. It states there is negative *privilege* on *obj* for *sub*.
neg_far$(sub, right(-, priv, obj), step)$, where *sub* is of subject entity sort. It is similar with pos_far$(sub, right(+, priv, obj), step)$.

[2] Because *Smodels* does not provide *max* and *min* functions, we have extended *Smodels* by adding them in it.

3.2.2 Propagation rules

We need the propagation rules because in most real world situations, the work to assign the authorization to all resources is burdensome and not necessary. The security officer prefers to assign them partially and propagate them to all resources based on propagation policy. In $\mathcal{L}_{Ans}$, we have propagation rules based on the relationships between objects or privileges as follows.

$$below(A_1, A_3) \leftarrow\ below(A_1, A_2),\ below(A_2, A_3) \quad (1)$$

Rule (1) is for the structured data propagation.

3.2.3 Authorization rules

Using negation as failure, if there is only positive authorization and no negative authorization, then we will conclude the positive authorization, and *vice versa*. On the other hand, if there are the positive and negative authorizations at the same time, we leave the decision problem to conflict resolution and decision policy. The following is our authorization rules.

$$exist\_pos\ \ (X, right(+, P, O)) \leftarrow auth(local, X, right(+, P, O), T). \quad (2)$$

$$exist\_neg\ \ (X, right(-, P, O)) \leftarrow auth(local, X, right(-, P, O), T). \quad (3)$$

Rules (2) and (3) denote that there are positive and negative authorization in the system respectively.

$$grant\ \ (X, right(+, access, O)) \leftarrow auth(local, X, right(+, P, O), T), not\ exist\_neg(X, right(-, P, O)). \quad (4)$$

Rule (4) makes positive authorization decision for the single subject request if there is only positive authorization and no negative authorization in the system.

$$grant\ \ (X, right(-, P, O)) \leftarrow not\ exist\_pos(X, right(+, P, O)). \quad (5)$$

Rule (5) makes negative authorization decision for the single subject request if there is no positive authorization, no matter whether there is positive authorization or not.

$$ggrant\ \ (L, right(+, P, O)) \leftarrow \\ auth(local, L, right(+, P, O), T), \\ match(L, right(+, P, O)), \\ not\ exist\_neg(L, right(-, P, O)). \quad (6)$$

Rule (6) makes positive authorization decision for the group subject request if there are only positive authorization and no negative authorization in the system, and the requesters satisfy the group subject requirement.

$$ggrant\ \ (L, right(-, P, O)) \leftarrow \\ not\ exist\_pos(L, right(+, P, O)). \quad (7)$$

Rule (7) makes negative authorization decision for the group subject request if there are group subject requests and no positive authorization decision that has been made.

3.2.4 Conflict resolution and decision rules

In an access control system, when both positive and negative authorization are permitted, the conflict occur. Most existing approaches deal with conflicts in the following ways:(1) No conflict policy. It relies on the security administrator to write the consistent authorization rules. If there are conflicts, errors happen[28]. (2) A fixed-conflict resolving policy based on relative authorization or specification. As pointed out in [23], this kind of policies include negative(positive)-take-precedence, strong and weak authorization, specific-take-precedence, and time-take-precedence. Moreover, Ruan et al [23, **?**] and Agudo et al [1], have proposed graph-based schemes to deal with distributed authorization, in which they present predecessor-take-precedence and strict-predecessor-take-precedence, respectively. It should be noted that the work in [1] is to generalize the proposal in [24] and can resolve more conflicts that are incomparable in [24]. (3) Flexible scheme to support multiple conflict resolving policies [15]. Logic-based approaches for distributed authorization can easily specify different policies that coexist in the same framework.

Our work is logic-based approach for distributed authorization, then it is feasible to integrate different conflict resolving policies into our approach. Comparing with the weighted-graph based approach [1, 24], we should mention that, it is easy to extend our language to handle weighted authorization because *Smodels* already provided weight literal representation in logic programming. In this paper, we choose trust-take-precedence policy, similar to the work in [23], to deal with the conflicts. We consider *delegation* as an action and assign the step for each authorization which is decided by the delegation step. All the authorizations arise from *local* originally. The step number denotes how far the authorization is away from *local* which reflects the trust extent. The smaller the authorization step, the more trust there is on this authorization. For this reason, we take preference to the smallest step authorization. If the conflict occurs with the same step, we deny the request. The following is the rules of our conflict resolution and decision policy.

$$pos\_far(X, right(+, P, O), T_1) \leftarrow \\ auth(local, X, right(+, P, O), T_1), \\ auth(local, X, right(-, P, O), T_2), \\ T_1 > T_2. \quad (8)$$

$$neg\_far(X, right(-, P, O), T_1) \leftarrow \\ auth(local, X, right(-, P, O), T_1), \\ auth(local, X, right(+, P, O), T_2), \\ T_1 > T_2. \quad (9)$$

Rule (8) denotes that for a positive authorization with step T_1, there is a negative authorization which has a smaller step. Rule(9) is *vice verse*.

$$grant\ \ (X, right(+, P, O)) \leftarrow \\ auth(local, X, right(-, P, O), T_2), \\ neg\_far(X, right(-, P, O), T_2), \\ auth(local, X, right(+, P, O), T_1), \\ not\ pos\_far(X, right(+, P, O), T_1). \quad (10)$$

Rule (10) makes positive authorization decision for the single subject request if there are both positive authorization and negative authorization, and a positive authorization with smallest step exists in the system.

$$grant\ \ (X, right(-, P, O)) \leftarrow \\ auth(local, X, right(+, P, O), T_1), \\ auth(local, X, right(-, P, O), T), \\ not\ neg\_far(X, right(-, P, O), T). \quad (11)$$

Rule (11) makes negative authorization decision for the single subject request if there are both positive authorization and negative authorization, and a negative authorization with smallest step exists or a negative authorization and a positive authorization have the same step which is smallest in the system.

$$ggrant\ \ (L, right(+, P, O)) \leftarrow \\ auth(local, L, right(-, P, O), T_2), \\ neg\_far(L, right(-, P, O), T_2), \\ match(L, right(+, P, O)), \\ auth(local, L, right(+, P, O), T_1), \\ not\ pos\_far(L, right(+, P, O), T_1). \quad (12)$$

$$ggrant\ \ (L, right(-, P, O)) \leftarrow \\ auth(local, L, right(+, P, O), T_1), \\ auth(local, L, right(-, P, O), T_2), \\ match(L, right(-, P, O)), \\ not\ neg\_far(L, right(-, P, O), T_2). \quad (13)$$

Rules (12) and (13) make positive and negative authorization decisions for the group subject requests respectively and have similar meaning with rules (10) and (11).

3.3 Transformation from $\mathcal{AL}$ to $\mathcal{L}_{Ans}$

As shown earlier, a rule $r_{\mathcal{D}}$ in the domain description $\mathcal{D}_{\mathcal{AL}}$ is of the following form

$$h_0 \ if \ b_1,\ b_2, \ldots, b_m,\ with\ absence\ b_{m+1}, \ldots,\ b_n. \quad (14)$$

where h_0 is the *head statement* denoted by $head(r_{\mathcal{D}})$ and b_is are *body statements* denoted by $body(r_{\mathcal{D}})$. We call the set of statements $\{b_1,\ b_2, \ldots, b_m\}$ *positive body statements*, denoted by $pos(r_{\mathcal{D}})$, and the set of statements $\{b_{m+1},\ b_{m+2}, \ldots, b_n\}$ *negative body statements*, denoted by $neg(r_{\mathcal{D}})$. If there is no confusion in context, we use *positive statements* and *negative statements* to express them respectively. In (14), if $m = 0$ and $n = 0$, the rule simply becomes h_0 and is called a *fact*.

In the next subsections we provide translation functions for $\mathcal{D}_{\mathcal{AL}}$ and $\mathcal{Q}_{\mathcal{AL}}$. The function $TansRules(\mathcal{D}_{\mathcal{AL}})$ translates the rules in the domain description $\mathcal{D}_{\mathcal{AL}}$ into an answer set program $\mathcal{P}$. We divide the process into three phases, body translation(see subsection 3.3.1), head translation(see subsection 3.3.2), and adding related rules (see subsections 3.2.2, 3.2.3, and 3.2.4). For query in language $\mathcal{AL}$, we provide $TransQuery(\mathcal{Q}_{\mathcal{AL}})$ to translate it into a program Π and ground literals $\varphi(+)$ and $\varphi(-)$.

In language $\mathcal{AL}$, there are function symbols, *assert-atom* and *auth-atom*. Correspondingly there are functions $exp(a_1, \ldots, a_n)$ and $right(sign, priv, obj)$ in language $\mathcal{L}_{Ans}$. In our translation, if there is no confusion in the context, we use *exp* and *right* to denote them in both languages.

3.3.1 Body transformation

In language $\mathcal{AL}$, there are four types of body statements, *relation statement, assert statement, delegation statement*, and *auth statement*. As delegation statement and auth statement have similar structure, we give their transformations together. For each rule $r_{\mathcal{D}}$, its body statement b_i is one of the following cases.

1. Relation statement:
   local says $below(arg_1,\ arg_2)$
   local says $neq(arg_1,\ arg_2)$
   local says $eq(arg_1,\ arg_2)$
   Replace them respectively in program $\mathcal{P}$ using:
   $$below(arg_1,\ arg_2). \quad (15)$$
   $$neq(arg_1,\ arg_2). \quad (16)$$
   $$eq(arg_1,\ arg_2). \quad (17)$$
   Where arg_1 and arg_2 in $below(arg_1, arg_2)$ are of *object or privilege entity sort*, arg_1 and arg_2 in $neg(arg_1, arg_2)$ and $eq(arg_1, arg_2)$ are of same type entity sort to specify they are equal or not equal. In *Smodels*, *neq* and *eq* are internal function and work as a constraint for the variables in the rules.
2. Assert body statement:
   issuer asserts *exp*.
   Replace it in program $\mathcal{P}$ using
   $$assert(issuer, exp) \quad (18)$$
   where *issuer* is a subject constant or variable, and *exp* is an assert atom.
3. Delegation body statement or auth body statement:
   issuer delegates *right* with depth k to *delegatee*
   issuer grants *right* to *grantee*.
   If *issuer* is a subject constant or variable, we replace the statements in program $\mathcal{P}$ using
   $$delegate(issuer, delegatee, right, k, T) \quad (19)$$
   $$auth(issuer, grantee, right, T), \quad (20)$$
   where k is delegation depth, T a step variable that means how many steps the delegation/right has gone through from *issuer* to *delegatee*/*grantee*.

We translate the positive statements as above, and for the negative body statements, we do the same translation and just add *not* before them.

3.3.2 Head transformation

In language $\mathcal{AL}$, there are also four types of head statements, *relation statement, assert statement, delegation statement*, and *auth statement*. If the head statement h_0 is a *relation statement* or an *assert statement*, the translations are same as the body statements. We adopt the rules (15), (18) to translate them respectively. In relation head statements, there are no statements for atom *neq* and *eq* that just be used as a variable constraints in body statements. Here we present the translation for *auth head statement*, and *delegation head statement*.

1. Auth head statement:
   issuer grants *right* to *grantee*

   If *grantee* is a subject constant or variable, we replace it by
   $$auth(issuer, grantee, right(Sn, P, O), 1) \quad (21)$$
   where 1 means the *right* is granted from *issuer* to *grantee* directly.
   We further add the following propagation rules for it:
   $$auth\ (\ issuer, grantee, right(Sn, P_1, O), 1) \leftarrow auth(issuer, grantee, right(Sn, P, O), 1), below(P_1, P).$$

$$auth\ (\ issuer, grantee, right(Sn, P, O_1), 1) \leftarrow auth(issuer, grantee, right(Sn, P, O), 1), below(O_1, O).$$

If *grantee* is a complex structure, *subject set, threshold*, or *subject extent set*, we introduce group subject entity l_{new} to denote the subjects in complex subject structures, and replace its head in program $\mathcal{P}$ as follows

$$auth(issuer, l_{new}, right(Sn, P, O), 1) \quad (22)$$

We also need to add the propagation rules similar to the above ones and the following different rules for different structures.

case 1: l_{new} is $[s_1, \ldots, s_n]$
$match(l_{new}, right) \leftarrow$
$auth(issuer, l_{new}, right, 1)$,
$n\{req(s_1, right), \ldots, req(s_n, right)\}n.$

case 2: l_{new} is $sthd(k, [s_1, s_2, \ldots, s_n])$
$match(l_{new}, right) \leftarrow$
$auth(issuer, l_{new}, right, 1)$,
$k\{req(s_1, right), \ldots,\ req(s_n, right)\}k.$

case 3: l_{new} is $dthd\ (k, S, sub\ assert\ exp(S)\ )$
$match(l_{new}, right) \leftarrow$
$auth(issuer, l_{new}, right, 1)$,
$k\{req(S, right):\ assert(sub, exp(S))\ \}k.$

case 4: l_{new} is $[dthd\ (k_1, S_1, s_1\ assert\ exp_1(S_1)\ ), \ldots, dthd\ (k_n, S_n, s_n\ assert\ exp_n(S_n)\ )].$
$match(l_{new}, right) \leftarrow$
$auth(issuer, l_{new}, right, 1)$,
$k_1\{req(S_1, right) : assert(s_1, exp_1(S_1))\}k_1$,
$\vdots$
$k_n\{req(S_n, right) : assert(s_n, exp_n(S_n))\}k_n.$

2. Delegation head statement:
issuer delegates *right* with depth k to *delegatee*
If *delegatee* is a subject constant or variable, we replace the statement in program P using

$$delegate(issuer, delegatee, right, k, 1) \quad (23)$$

where k is the delegation depth, and 1 means the *issuer* delegates the *right* to *delegatee* directly.
Moreover, we need to add the following implied rules for it in program $\mathcal{P}$:
Prop-delegation rules: Based on the structured resources, the delegation can be propagated as following rules.

$$delegate\ \ (issuer, delegatee, right(\Box, P_1, O), k, 1) \leftarrow delegate(issuer, grantee, right(\Box, P, O), k, 1), below(P_1, P).$$

$$delegate\ \ (issuer, delegatee, right(\Box, P, O_1), k, 1) \leftarrow delegate(issuer, delegatee, right(\Box, P, O), k, 1), below(O_1, O).$$

Auth-delegation rule: When the issuer delegates a right to the delegatee, the issuer will agree with the delegatee to grant the right to other subjects within delegation depth. The authorization step increases 1.
$auth(issuer, S, right(Sn, P, O), T + 1) \leftarrow$
$delegate(issuer, delegatee, right(\Box, P, O), k, 1)$,
$auth(delegatee, S, right(Sn, P, O), T).$

Delegation-chain rule: The delegation can be redelegated within delegation depth.
$delegate(issuer, S, right(\Box, P, O),$
$min(k\text{-}Step, Dep), 1 + T) \leftarrow$
$delegate(issuer, delegatee, right(\Box, P, O), k, 1)$,
$delegate(delegatee, S, right(\Box, P, O), Dep, T)$,
$T\ <\ k.$

Self-delegation rule: The delegatee can delegate the right to himself/herself within k depth.

$delegate(delegatee, delegatee, right, Dep, 1) \leftarrow$
$delegate(issuer, delegatee, right, k, 1)$,
$Dep \leq k.$

Weak-delegation rule: If there is a delegation with k steps, we can get the delegation with steps less than k.
$delegate(issuer, delegatee, right, Dep, 1) \leftarrow$
$delegate(issuer, delegatee, right, k, 1)$,
$Dep < k.$

If *delegatee* is a complex structure, *subject set, static threshold*, or *dynamic threshold*, we introduce a new group subject l_{new} to denote the subjects in complex structures, and replace the statement in program $\mathcal{P}$ using
$delegate(issuer, l_{new}, right, k, 1).$
We also need to add additional rules for it. Because there are similar rules for different complex *delegatee* structure, here we just present the rules for *subject set* structure.
Prop-delegation rules: Based on the structured resources, the delegation can be propagated similar with those for single delegatee.
$delegate(issuer, l_{new}, right(\Box, P_1, O), k, 1) \leftarrow$
$delegate(issuer, l_{new}, right(\Box, P, O), k, 1)$,
$below(P_1, P).$
$delegate(issuer, l_{new}, right(\Box, P, O_1), k, 1) \leftarrow$
$auth(issuer, l_{new}, right(\Box, P, O), k, 1)$,
$below(O_1, O).$

Auth-delegation rule: If an issuer delegates a right to a group subject, and all the members in the group authorize this right to a subject, then the issuer agree with this authorization. The new authorization step is 1 plus the biggest one among the group authorizations because the trust for the new authorization is less than any group authorizations.
$auth(issuer, S, right, T + 1) \leftarrow$
$delegate(issuer, l_{new}, right, k, 1)$,
$auth(s_1, S, right, T_1)$,
$\vdots$
$auth(s_n, S, right, T_n)$,
$T = max(T_1, \ldots, T_n).$

Delegation-chain rule: If an issuer delegates a right to a group subject, and all the members in the group re-delegate this right to a subject, then the issuer agree with this re-delegation. The new delegation depth is the smallest one among k minus $step_i$s and Dep_is and the new delegation step is 1 plus the biggest one among the group delegations.

$$
\begin{aligned}
&delegate(issuer, S, right, T_1, T_2+1) \leftarrow \\
&\quad delegate(issuer, l_{new}, right, k, 1), \\
&\quad delegate(s_1, S, right, Dep_1, Step_1), \\
&\qquad \vdots \\
&\quad delegate(s_n, S, right, Dep_n, Step_n), \\
&\quad T_1 = min(k\text{-}Step_1, \ldots, k\text{-}Step_n, Dep_1, \ldots, Dep_n), \\
&\quad T_2 = max(Step_1, \ldots, Step_n), \\
&\quad T_1 > 0.
\end{aligned}
$$

3.3.3 Query Transformation

In language $\mathcal{AL}$, there are two kinds of queries, single subject query and group subject query. We present the function $TransQuery(\mathcal{Q}_{\mathcal{AL}})$ for both of them and this function returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$.

If $\mathcal{Q}_{\mathcal{AL}}$ is a single subject query,

s *requests* $right(+, p, o)$,

TransQuery returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$ as follows respectively,

$\{req(s, right(+, p, o))\}$,
$grant(s, right(+, p, o))$, and
$grant(s, right(-, p, o))$.

If $\mathcal{Q}_{\mathcal{AL}}$ is a group subject query,

$[s_1, s_2, \ldots, s_n]$ *requests* $right(+, p, o)$.

TransQuery returns program Π and ground literals $\varphi(+)$ and $\varphi(-)$ as follows respectively,

$\{req(\ s_i, right(+, p, o)) \mid i = 1, \ldots, n\ \}$,
$ggrant(l, right(+, p, o)), and$
$ggrant(l, right(-, p, o))$,

where l is a group subject entity to denote the set of subjects, $[s_1, \ldots, s_n]$.

4 Scenarios

In this section we represent two specific authorization scenarios to demonstrate the features of language $\mathcal{AL}$.

Scenario 1 A company chooses to have multiparty control for emergency key recovery. If a key needs to be recovered, three persons are required to present their individual PINs. They must be from different departments: a member of management, an individual from auditing, and one individual from IT department. The system trusts the manager of Human Resource Department to identify the staff of the company. The domain description $\mathcal{D}_{\mathcal{AL}}$ for this scenario then consists of the following rules represented using language $\mathcal{AL}$.

$$
\begin{aligned}
&local \text{ grants } right(+, recover, key) \text{ to} \\
&\quad [\, dthreshold(1, X, hrM \text{ asserts } isAManager(X)), \\
&\quad\ dthreshold(1, Y, hrM \text{ asserts } isAnAuditor(Y)), \\
&\quad\ dthreshold(1, Z, hrM \text{ asserts } isATech(Z))\,]. \\
&hrM \text{ asserts } isAManager(alice). \\
&hrM \text{ asserts } isAnAuditor(bob). \\
&hrM \text{ asserts } isAnAuditor(carol). \\
&hrM \text{ asserts } isATech(david).
\end{aligned}
$$

We translate them into language $\mathcal{L}_{Ans}$,

$$
\begin{aligned}
&auth(local, l, right(+, recovery, key), 1). \\
&match(l, right(+, recovery, key)) \leftarrow \\
&\quad auth(local, l, right(+, recovery, key), 1), \\
&\quad 1\{req(X, right(+, recovery, key)) : \\
&\qquad\qquad assert(hrM, isAManager(X))\}1, \\
&\quad 1\{req(Y, right(+, recovery, key)) : \\
&\qquad\qquad assert(hrM, isAnAuditor(Y))\}1, \\
&\quad 1\{req(Z, right(+, recovery, key)) : \\
&\qquad\qquad assert(hrM, isATech(Z))\}1. \\
&assert(hrM, isAManager(alice)). \\
&assert(hrM, isAnAuditor(bob)). \\
&assert(hrM, isAnAuditor(carol)). \\
&assert(hrM, isATech(david)).
\end{aligned}
$$

In this scenario, the program $\mathcal{P}$ consists of the above translated rules, and those authorization rules we specified in section 3.2.3. If Alice, Bob, and David make a request to recover a key together, that is,

$[alice, bob, david]$ requests $right(+, recovery, key)$.

After translation, we get program Π,

$req(alice, right(+, recovery, key))$,
$req(bob, right(+, recovery, key))$,
$req(david, right(+, recovery, key))$,

and the ground literal $\varphi(+)$ is,

$ggrant(l, right(+, recovery, key))$.

where l is a group subject entity to represent the set of subjects, $[alice, bob, david]$.

Then program $\mathcal{P} \cup \Pi$ (Refer to the Appendix A for complete program) has only one answer set, and $ggrant(l, right(+, recovery, key))$ is in the answer set. So $(\mathcal{P} \cup \Pi) \models ggrant(l, right(+, recovery, key))$. That is the request is permitted.

Now if we consider that Alice, Bob, and Carol make the same request, the rule for $match(l, right(+, recovery, key))$ can not be satisfied. From the authorization rules (6) and (7) in section 3.2.3, the system deny the request from Alice, Bob, and Carol. A complete ASP program representing this scenario is given in Appendix A.

Scenario 2 A server provides the services including *http*, *ftp*, *mysql*, and *smtp*. It sets up a group for them, called *services*. The server delegated the right of assigning all the services to the security officer *so* with depth 3. The security officer *so* grants the services to *staff*. The service *mysql* can not be accessed if the staff is on holidays. Officer *so* can get information of staff from the human resource manager *hrM*. The policies and credentials are described using language $\mathcal{AL}$ as follows.

$local\ says\ below(http, services)$.
$local\ says\ below(ftp, services)$.
$local\ says\ below(mysql, services)$.
$local\ says\ below(smtp, services)$.
$local$ delegates $right(\Box, access, services)$
  with depth 3 to so.
so grants $right(+, access, Y)$ to X
  if hrM asserts $isStaff(X)$,
  $local\ says\ below(Y, services)$,
  $local\ says\ neq(Y, mysql)$.
so grants $right(+, access, mysql)$ to X
  if hrM asserts $isStaff(X)$,
  $with\ absence\ hrM$ asserts $onHoliday(X)$.
hrM asserts $isStaff(alice)$.
hrM asserts $isStaff(bob)$.
hrM asserts $onHoliday(alice)$.

For this scenario, we give the complete $\mathcal{L}_{Ans}$ program in Appendix B. Through *Smodels*, we get one and only one answer set. Within this answer set, we can extract the authorization path for Alice as follows:

$auth(local, alice, right(+, access, http), 2)$
$auth(so, alice, right(+, access, http), 1)$
$grant(alice, right(+, access, http))$
$grant(alice, right(-, access, mysql))$

The predicates *auth* are helpful for us to find the authorization path. We can find that the authorization passes from *local* to *so*, and then from *so* to *alice*. In many applications, such authorization path and related delegation chain play an essential role to identify the validity of requests [9,18].

5 Computational Properties

In this section, we study basic computational properties of language $\mathcal{AL}$. In language $\mathcal{AL}$, the authorization policy and associated delegations for a specific problem domain is represented as a domain description $\mathcal{D}_{\mathcal{AL}}$, which is a finite set of rules which is a policy base including local policy and credentials from all trusted entities. Then for each access request to the resource, the decision is made on the basis of reasoning mechanism underlying the framework we developed earlier.

From Definition 3, we can see that given a domain description $\mathcal{D}_{\mathcal{AL}}$ and a query $\mathcal{Q}_{AL}$ (see Figure 1 for the syntax of a query), there are three steps to answer this query: (1) transfer $\mathcal{D}_{\mathcal{AL}}$ and $\mathcal{Q}_{AL}$ to a logic program $\mathcal{T}$; (2) computer the answer sets of $\mathcal{T}$; (3)check if $grant(s, right(sn, p, o))$ or $ggrant(l, right(sn, p, o))$ is in all answer sets of $\mathcal{T}$. Step 1 is achieved through two transformation functions: $TransRules(\mathcal{D}_{\mathcal{AL}}) = \mathcal{P}$ and $TransQuery(\mathcal{Q}_{\mathcal{AL}}) = <\Pi, \varphi(+), \varphi(-)>$. That is, $\mathcal{T} = \mathcal{P} \cup \Pi$. For step 2, we provide function $stable(\mathcal{T})$ which returns all answer sets of program $\mathcal{T}$. Step 3 is just a simple checking that can be done in linear time. So the main computational cost for our approach is on Steps 1 and 2. The following proposition presents the complexity result for achieving Step 1.

Proposition 1 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description of language $\mathcal{AL}$ and $\mathcal{Q}_{\mathcal{AL}}$ a query. Then $TRanRules(\mathcal{D}_{\mathcal{AL}})$ can be computed in time $\mathcal{O}(|\mathcal{D}_{\mathcal{AL}}|)$ and $TransQuery(\mathcal{Q}_{AL})$ can be computed in time $\mathcal{O}(|\mathcal{Q}_{AL}|)$*(here $|\mathcal{D}_{\mathcal{AL}}|$ is the size of $\mathcal{D}_{\mathcal{AL}}$ and $|\mathcal{Q}_{AL}|$ is the length of $\mathcal{Q}_{AL}$.).

Proof From the transformation process, it is clear that $TransQuery(\mathcal{Q}_{\mathcal{AL}})$ entirely depends on the query formula's length. So it can be obtained in linear time in terms of $\mathcal{Q}_{\mathcal{AL}}$'s size. On the other hand, rule transformation includes body transformation and head transformation. After body transformation, the number of rules does not change. So $|\mathcal{P}_{body}| = |\mathcal{D}_{\mathcal{AL}}|$. During head transformation, for the complex structure in auth head and delegation head transformation, $|\mathcal{P}_{head}| = c|\mathcal{D}_{\mathcal{AL}}|$, where c is a constant number. So we conclude that $|\mathcal{P}_{head}| = \mathcal{O}(|\mathcal{D}_{\mathcal{AL}}|)$.

Now we consider the computation of Step 2. For stable($\mathcal{T}$), we use Smodels to compute the answer sets of logic programs. It is well known that deciding whether a program has an answer set is NP-complete [3]. Consequently, Smodels usually needs exponential time to compute a program's answer sets. Therefore, it is important to identify proper subclasses of the authorization domains where queries can be answered in polynomial time. In the following section, we will define two subclasses of language $\mathcal{AL}$ in which queries can be computed in polynomial time.

The domain description of language $\mathcal{AL}$ includes infinite rules and the basic unit of a rule is a statement. We have four types of statements, *relation statement, assertion statement, delegation statement*, and *auth statement*. To simplify our investigation, we consider each statement as a predicate with n terms, $p(t_1, t_2, \ldots, t_n)$ in which p denotes the statement type, and t_is are terms to denote the variable parts in the statement. The four types of statements have the following forms,

$RelStmt(issuer, relAtomName, atomArg1, atomArg2)$
$AssertStmt(issuer, assertAtomName, atomArg1, \ldots,$
  $atomArgn)$
$DelegationStmt(issuer, receiver, step, priv, obj)$
$AuthStmt(issuer, receiver, sign, priv, obj)$

For instance, we denote a relation statement, *"local says below(alice, postgraduate)"* using predicate form, $RelStmt(local, below, alice, postgraduate)$. In such predicate presentation, each term has a type which can be subject, subject structure, object, privilege, sign, integer, relation atom name, or assert atom name. Subject structures are special terms and have four types: subject set, subject static threshold, subject dynamic threshold and subject extended dynamic threshold. *Subject set* and *subject static threshold* have static subject pools, while *subject dynamic threshold* and *subject extended dynamic threshold* have dynamic subject pools. In the static subject pool $[s_1, s_2, \ldots, s_n]$, each s_i is a *member of the static subject pool*. For a dynamic subject pool, each constant subject in the domain of the application system can be a *member of the dynamic subject pool*.

Definition 4 Term t_1, and t_2 are *compatible*, denoted by $t_1 \simeq t_2$, if t_1 and t_2 are same type terms, and one of the following conditions holds:

1. t_1 and t_2 are constant terms with the same name;
2. at least one of t_1 and t_2 is a variable term; or
3. t_1 is a subject constant, t_2 is a subject structure, and t_1 is a member of t_2.

For example, if term t_1 is a subject *alice*, t_2 is a subject variable S, t_3 is a static threshold $sth(2, [bob, carol, david])$, and t_4 is a dynamic threshold $dth(3, S, hrM\ asserts\ isAStaff(S))$, we say (t_1, t_2), (t_1, t_4), (t_2, t_3), and (t_2, t_4) are compatible term pairs.

Definition 5 Two statements s_1, s_2 are *compatible*, denoted by $s_1 \simeq s_2$, if s_1 and s_2 have the predicate forms s_1' and s_2' respectively, and

1. s_1' and s_2' are the same type predicates,
2. all the corresponding terms of s_1' and s_2' are compatible.

From the above definitions, it is easy to see that a statement is compatible to itself.

Definition 6 Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description of language $\mathcal{AL}$ and r_p and r_q be two rules in $\mathcal{D}$. We define a set $\mathcal{S}(r_p)$ of statements with respect to r_p as follows:

$\mathcal{S}_0 = \{head(r_p)\}$;
$\mathcal{S}_i = \mathcal{S}_{i-1} \cup \{head(r) \mid head(r') \simeq s\ where\ s \in pos(r)\ and$
$r'\ are\ those\ rules\ such\ that\ head(r') \in \mathcal{S}_{i-1}\}$;
$\mathcal{S}(r_p) = \bigcup_{i=1}^{\infty} \mathcal{S}_i$.

We say that r_q is *defeasible through* r_p in $\mathcal{D}_{\mathcal{AL}}$ if and only if $neg(r_q) \cap^c \mathcal{S}(r_p) \neq \emptyset$ [3][4].

[3] $\cap^c$ is to get a compatible joint set of two statement sets. Formally, $A \cap^c B = \{s_1, s_2 | s_1 \simeq s_2,\ where\ s_1 \in A,\ and\ s_2 \in B$.

[4] See Section 3.3 for definitions of $head(r)$, $pos(r)$ and $neg(r)$ in language $\mathcal{AL}$.

Intuitively, if r_q is defeasible through r_p in $\mathcal{D}_{\mathcal{AL}}$, then there exists a sequence of rules $r_1, r_2, \ldots, r_l, \ldots$ such that $head(r_p)$ occurs in $pos(r_1)$, $head(r_i)$ occurs in $pos(r_{i+1})$ for all $i = 1, \ldots,$ and for some k, $head(r_k)$ occurs in $neg(r_q)$. Under this condition, it is clear that by triggering rule r_p in $\mathcal{D}_{\mathcal{AL}}$, it is possible to defeat rule r_q if rules $r_1, \ldots, r_k$ are triggered as well. As a special case that $\mathcal{S}(r_p) = \{head(r_p)\}$, r_q is defeasible through r_p iff $head(r_p) \in neg(r_q)$.

Definition 7 Given a domain description $\mathcal{D}_{\mathcal{AL}}$, we define its *defeasible graph* $\mathcal{DG} = \langle V, E \rangle$, where V is the set of rules r_i in $\mathcal{D}_{\mathcal{AL}}$ as the vertices and E the set of $\langle r_i, r_j \rangle$ which is a directed edge to denote r_j is defeasible through r_i.

Consider a simple example. Suppose $a, b, c, \ldots$ are statements , and $a', b', c', \ldots$ are their corresponding compatible statements in language $\mathcal{AL}$, and we have the following domain description $\mathcal{D}_{\mathcal{AL}}$:

$r_1:\ b\ if\ a.$
$r_2:\ c\ if\ b'.$
$r_3:\ d\ if\ with\ absence\ c.$
$r_4:\ e\ if\ with\ absence\ b.$

Based on the definitions above, we conclude that rule r_3 is defeasible through r_1 and r_2, and rule r_4 is defeasible through r_1. Then we have the following defeasible graph:

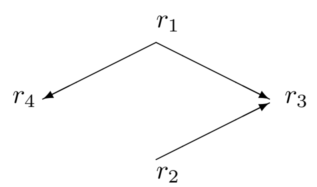

Figure 2: The defeasible graph

The logic program $\mathcal{P}$ with answer set semantics consists of finite set of rules. A rule r is expressed as follows:

$$L_0 \leftarrow L_1, \ldots, L_m,\ not\ L_{m+1}, \ldots,\ not\ L_n.$$

where each $L_i (0 \leq i \leq n)$ is a literal. The program $\mathcal{P}$ is *ground* if each rule in $\mathcal{P}$ is ground. Let r be a ground rule of the above form, we use $pos(r)$ to denote the set of literals in the body of r without negation as failure $\{L_1, \ldots, L_m\}$, $neg(r)$ to denote the set of literals in the body of r with negation as failure $\{L_{m+1}, \ldots, L_n\}$, and $body(r)$ to denote $pos(r) \cup neg(r)$. L_0 is called the head of the rule, denoted by $head(r)$. By extending these notations, we use $pos(\mathcal{P})$, $neg(\mathcal{P})$, $body(\mathcal{P})$, and $head(\mathcal{P})$ to denote the unions of corresponding components of all rules in program $\mathcal{P}$, e.g. $body(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} body(r)$.

We present the concepts of local stratification and call consistence for extended logic programs [3,29].

Definition 8 Let $\mathcal{P}$ be an extended logic program and Lit be the set of all ground literals of $\mathcal{P}$.

1. A *local stratification* for $\mathcal{P}$ is a function *stratum* from *Lit* to the countable ordinals.
2. Given a local stratification *stratum*, we extend it to ground literals with negation as failure by setting $stratum(not\ L) = stratum(L) + 1$, where L is a ground literal.
3. A rule $L_0 \leftarrow L_1, \ldots, L_m,\ not\ L_{m+1}, \ldots,\ not\ L_n$ in $\mathcal{P}$ is locally stratified with respect to *stratum* if
   $stratum(L_0) \geq stratum(L_i)$, where $1 \leq i \leq m$, and
   $stratum(L_0) > stratum(not\ L_j)$, where $m + 1 \leq j \leq n$.
4. $\mathcal{P}$ is called *locally stratified* with respect to *stratum* if all of its rules are locally stratified. $\mathcal{P}$ is called *locally stratified* if it is locally stratified with respect to some local stratification.

Definition 9 An extended program is said to be *call-consistent* if its dependency graph does not have a cycle with an odd number of negative edges.

Lemma 1 *Let P_1, P_2 be two locally stratified logic programs. Program $P_1 \cup P_2$ is locally stratified if $head(P_2) \cap body(P_1) = \emptyset$.*

Proof Because P_1 and P_2 are two locally stratified propositional logic programs, their dependency graphes D_{P_1} and D_{P_2} both do not contain any negative cycles. Now $head(P_2) \cap body(P_1) = \emptyset$. We assume program $P_1 \cup P_2$ is not locally stratified and its dependency graph $D_{P_1 \cup P_2}$ contains a cycle with at least one negative edge, denoted by $< a_1, a_2, \ldots, a_{i-1}^-, a_i, \ldots, a_n, a_1 >$, in which a_is are atoms, the atom sequence means there are edges in dependency graph to connect them one by one, a_{i-1}^- means there is a negative edge from a_{i-1} to a_i.

From the above condition, there are the following rules in program $P_1 \cup P_2$:

$r_1 :\ a_2 \leftarrow \ldots, a_1, \ldots$
$r_2 :\ a_3 \leftarrow \ldots, a_2, \ldots$
$\cdots$
$r_{i-1} :\ a_i \leftarrow \ldots, not\ a_{i-1}, \ldots$
$\cdots$
$r_{n-1} :\ a_n \leftarrow \ldots, a_{n-1}, \ldots$
$r_n :\ a_1 \leftarrow \ldots, a_n, \ldots$

In the above rules, at lease one is from program P_1. Assume r_1 is in program P_1, r_n should be in P_1 also, because $a_1 = head(r_n)$, $a_1 \in body(r_1)$ and $head(P_2) \cap body(P_1) = \emptyset$. For the same reason, we conclude that all of $r_{n-1}, r_{n-2}, \ldots, r_2$ should be in program P_1. This implies that P_1 is not locally stratified. The contradiction happens. Similarly, if we assume r_1 is in program P_2, we will conclude that P_2 is not locally stratified. So we prove that if P_1 and P_2 are two locally stratified logic programs and $head(P_2) \cap body(P_1) = \emptyset$, then the program $P_1 \cup P_2$ is locally stratified too.

Lemma 2 *Let P_1, P_2 be two call consistent logic programs. Program $P_1 \cup P_2$ is call consistent if $head(P_2) \cap body(P_1) = \emptyset$.*

Proof The dependency graphes of program P_1 and P_2 do not have a cycle with an odd number of negative edges because P_1 and P_2 are call consistent. Now $head(P_2) \cap body(P_1) = \emptyset$. Suppose program $P_1 \cup P_2$ is not call consistent and its dependency graph has a cycle with an odd number of negative edges. We can construct an atom sequence $< a_1, a_2, \ldots, a_n, a_1 >$ to denote a cycle with an odd number of negative edges, where a_is are atoms and the sequence means there are positive or negative edges to connect the atoms one by one. We get the following rules in program $P_1 \cup P_2$:

$r_1 :\ a_2 \leftarrow \ldots,\ [not]^5\ a_1, \ldots$
$r_2 :\ a_3 \leftarrow \ldots,\ [not]\ a_2, \ldots$
$\vdots$
$r_{i-1} :\ a_i \leftarrow \ldots,\ [not]\ a_{i+1}, \ldots$
$\vdots$
$r_{n-1} :\ a_n \leftarrow \ldots,\ [not]\ a_{n-1}, \ldots$
$r_n :\ a_1 \leftarrow \ldots,\ [not]\ a_n, \ldots$

In the above rules, at least one is from program P_1. We assume r_1 is in program P_1. Because $head(P_2) \cap body(P_1) = \emptyset$, r_n should be in program P_1 as well. For the same reason, we conclude that all of $r_{n-1}, r_{n-2}, \ldots, r_2$ should be in program P_1. This implies that P_1 is not a call consistent program. The contradiction happens. Similarly, if we assume r_1 is in program P_2, we will conclude that P_2 is not a call consistent program. This improves our result.

Lemma 3 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description and $\mathcal{P}$ the translated logic program in language $\mathcal{L}_{Ans}$. If the defeasible graph $\mathcal{DG}$ of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle, then $\mathcal{P}$ is* locally stratified.

Proof Suppose $\mathcal{D}_{\mathcal{AL}}$ is a domain description of language $\mathcal{AL}$ and its defeasible graph does not have a cycle.

The semantics of language $\mathcal{AL}$ is to translate $\mathcal{D}_{\mathcal{AL}}$ into a logic program $\mathcal{P}$. The process includes three steps: (a) translate rules in $\mathcal{D}_{\mathcal{AL}}$ into logic program rules and obtain the logic program $\mathcal{P}_1'$; (b) add the propagation rule (1) into program $\mathcal{P}_1'$ and get logic program $\mathcal{P}_1$; (c) get logic program $\mathcal{P}_2$ which consists of authorization rules and conflict decision rules (refer to Section 3.2.3 and 3.2.4) and $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$.

The basic unit of rules in $\mathcal{D}_{\mathcal{AL}}$ is a statement which has a corespondent predicate in logic program. From the translation process (refer to section 3.3), we have the following observation:

[5] [not] means 'not' is an option to denote that the following atom is positive or negative.

Observation: if the defeasible graph of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle, then the dependency graph of program $\mathcal{P}_1'$ does not have a negative cycle. So program $\mathcal{P}_1'$ is locally stratified.[6]

Consider the program $\mathcal{P}_1$ which is program $\mathcal{P}_1'$ plus the propagation rule:

$below(A_1, A_3) \leftarrow below(A_1, A_2),\ below(A_2, A_3).$

Because in the domain description, the relation statements related with *below* just are facts about resource relationship and as conditions for authorization rules, the program $\mathcal{P}_1$ is also locally stratified.

From the observation of authorization rules and conflict resolving rules in section 3.2.3 and section 3.2.4, obviously $\mathcal{P}_2$ is locally stratified and $head(\mathcal{P}_2) \cap body(\mathcal{P}_1) = \emptyset$. From Lemma 1, we conclude that the logic program $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ is locally stratified.

Lemma 4 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description and $\mathcal{P}$ the translated logic program in language $\mathcal{L}_{Ans}$. If the defeasible graph $\mathcal{DG}$ of $\mathcal{D}_{\mathcal{AL}}$ does not have a cycle with an odd number edges, then $\mathcal{P}$ is* call consistent.

Proof The Lemma 4 can be proved in a similar way of that in Lemma 3.

Theorem 1 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description. If its* defeasible graph $\mathcal{DG}$ *does not have a cycle, then $\mathcal{D}_{\mathcal{AL}}$ has a unique model that can be computed in polynomial time.*

Proof From [3], a locally stratified AnsProlog program has the unique answer set and can be computed in polynomial time. Based on Lemma 3 and the definition of semantics of domain description $\mathcal{D}_{\mathcal{AL}}$, we can prove the result.

Theorem 2 *Let $\mathcal{D}_{\mathcal{AL}}$ be a domain description. If its* defeasible graph $\mathcal{DG}$ *does not have a cycle with an odd number edges, then $\mathcal{D}_{\mathcal{AL}}$ has at lease one model that can be computed in polynomial time.*

Proof From [3], a call consistent AnsProlog program has at lease one answer set and can be computed in polynomial time. Based on Lemma 4 and the definition of semantics of domain description $\mathcal{D}_{\mathcal{AL}}$, we can prove the result.

6 Related Work and Conclusion

In this paper, we developed an authorization language $\mathcal{AL}$ to specify distributed authorization with delegation. We used Answer Set Programming as a foundational basis for its semantics and computation. As we have showed, $\mathcal{AL}$ has a rich expressive power representing not only nonmonotonic policies and positive and negative authorization, but also structured resources and privileges, partial authorization and delegation, and separation of duty policies.

As we indicated earlier, our formulation has implementation advantages due to recent development of Answer Set Programming technology in AI community[7], where many existing approaches do not have. The both scenarios in section 4 have been fully implemented through Answer Set Programming.

We also investigated the computational issue related to language $\mathcal{AL}$. Due to the intractability of answer set programming, in our formulation, we dealt with this problem in two ways. One way is to employ the state of the art technology of Answer Set Programming to develop optimization strategies to improve the computation process for query evaluation. We applied *lparse* to ground and simplify the logic programs [21], which is a default front-end to *Smodels*. The other way is to identify more general tractable classes of $\mathcal{AL}$ domains by applying some computational results in logic programs. We considered that when an extended logic program is *locally stratified* or *call-consistent*, then this program must have an answer set, and such answer set can be computed in polynomial time. By examining proper conditions, we identified two classes of $\mathcal{AL}$ domains, for which their $\mathcal{L}_{Ans}$ translation will always be locally stratified or call-consistent. In this way, any query under those types of domains can be evaluated in polynomial time.

Our approach developed in this paper has been implemented under the application domain of XML based resource management. The features of structured resources and partial delegation and authorization are suitable to specify delegable authorization for fine-grained XML resources. The detailed system structure and algorithms will be described in our another paper.

Delegation is an important feature that distinguishes distributed authorization from traditional centralized authorization. Some approaches use logic to specify this problem [19, 20, 25], while other approaches use graph representation [1, 24, 23]. As we pointed out previously, [19] and [20] do not express the nonmonotonic policies which is important for distributed environment. Although *DAP* [25] has nonmonotonic features, it can not express the complex policies such as delegation depth control, partial delegation, threshold structure and separation of duty. Also, unlike *DAP*, our $\mathcal{AL}$ is a high level formal langauge which is easier for the end-user to write a proper policy base. The graph-based approaches [1, 23, 24], on the other hand, indeed address delegation depth and conflict resolution issues, especially using weighted graph, however, they do not support the complex authorization and delegation representations such as separation of duty, threshold structure, and partial delega-

[6] The result can be proved directly from the definition of defeasible graph of $\mathcal{D}_{\mathcal{AL}}$ and dependency graph of program $\mathcal{P}_1'$.

[7] Please refer to http://www.tcs.hut.fi/Software/smodels/index.html

tion and authorization. Furthermore, it is not known yet whether these approaches have been implemented.

Our work presented in this paper can be further extended. One important topic is called *delegation chain discovery*. To answer an access request, our current approach will only generate a result to grant, deny, or be undecided to the request. However, very often, it is more useful to also explain why such request can be granted, denied or undecided. In a distributed environment, this could be difficult to achieve because the underlying delegation procedure may be very complex. Using Answer Set Programming, it is possible to retrieve such complex delegation chains from the answer sets that we have computed.

References

1. Agudo I., Lopez J., Montenegro J. A.(2005) A Representation Model of Trust Relationships with Delegation Extensions. In *Proceedings of the 3th International Conference on Trust Management*, pp 116-130.
2. Anger Chr., Konczak K., Linke Th., Schaub T.(2005) A glimpse of answer set programming. *Künstliche Intelligenz* 19(1): 12-17.
3. Baral C.(2003) *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.
4. Bertino E., Buccafurri F., Ferrari E., Rullo P.(1999) A logical framework for reasoning on data access control policies. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop(CSFW-12)*, pp 175-189, IEEE Computer Society Press, Los Alamitos, CA.
5. Blaze M., Feigenbaum J., Lacy J.(1996) Decentralized trust management. In *Proceedings of the Symposium on Security and Privacy*, pp 164-173, IEEE Computer Society Press, Los Alamitos.
6. Blaze M., Feigenbaum J., Strauss M.(1998) Compliance-checking in the policyMaker trust management system. In *Proceedings of Second International Conference on Financial Cryptography (FC'98)*, LNCS, Vol.1465, pp 254-274. Springer.
7. Blaze M., Feigenbaum J., Ioannidis J., Keromytis A. D.(1999) The role of trust Management in distributed systems. In *Secure Internet Programming*, LNCS, Vol.1603, pp 185-210, Springer, Berlin.
8. Blaze M., Feigenbaum J., Ioannidis J., Keromytis A. D.(1999) The keyNote trust-management system, Version 2, Internet Engineering Task Force RFC 2704. http://www.ietf.org/rfc/rfc2704.txt
9. Clarke D., Elien J., Ellison C., Fredette M., Morcos A., Rivest R. L.(1999) Certificate chain discovery in SPKI/SDSI, manuscript.
10. Elien J.(1998) Certificate discovery using SPKI/SDSI 2.0 certificates. Masters Thesis, MIT LCS. http://theory.lcs.mit.edu/ cis/theses/elien-masters.ps
11. Ellison C., Frantz B., Lampson B., Rivest R., Thomas B., Ylonen T. (1999) SPKI certificate theory. Internet Engineering Task Force RFC 2693. http://www.ietf.org/rfc/rfc2693.txt
12. Ellison C., Frantz B., Lampson B., Rivest R., Thomas B., Ylonen T.(1999) Simple public key certificate, Internet Draft.
13. Gelfond M., Lifschitz V.(1988) The stable model semantics for logic programming. *Logic Programming: Proc. of the Fifth Int'L Conf. and Symp.*, R. Kowalski and K. Bowen, editors, pp 1070-1080. MIT Press.
14. ITU-T Rec. X.509 (revised), The directory - authentication framework, International Telecommunication Union.
15. Jajodia S., Samarati P., Subrahmanian V. S.(2001) Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, Vol.26(2): 214-260.
16. Kent S. T.(1993) Internet privacy enhanced mail, *Communications of the ACM*, 36(8): 48-60.
17. Li N., Feigenbaum J., Grosof B. N.(1999) A logic-based knowledge representation for authorization with delegation (extended abstract). In *Proceedings of the IEEE Computer Security Foundations Workshop*, pp 162-174. IEEE Computer Society Press, Los Alamitos, CA.
18. Li N., Winsborough W. H., Mitchell J. C.(2003) Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1): 35-86.
19. Li N., Mitchell J. C., Winsborough W. H.(2002) Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp 114-130. IEEE Computer Society Press.
20. Li N., Grosof B. N., Feigenbaum J.(2003) Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, Vol 6(1): 128-171.
21. Niemela I., Simons P., Syrjanen T.(2000) Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-monotonic Reasoning.*
22. Rivest R. L., Lampson B.(1996) SDSI - A simple distributed security infrastructure. http://theory.lcs.mit.edu/ rivest/sdsi11.html
23. Ruan C., Varadharajan V.(2002) Resolving Conlicts in Authorization Delegations. In *Proceedings of the 7th Australian Conference on Information Security and Privacy*, pp 271-285, LNCS 2384.
24. Ruan C. and Varadharajan V.(2004) A Weighted Graph Approach to Authorization Delegation and Conflict Resolution. In *Proceedings of the 9th Australian Conference on Information Security and Privacy*, pp 402-413.
25. Ruan C., Varadharajan V.and Zhang Y.(2002) Logic-Based Reasoning on Delegatable Authorizations. In *Proceedings of the 13th International Symposium on Foundations of Intelligent Systems,* pp 185-193.
26. Syrjänen T. Lparse 1.0 User's Mannual. http://www.tcs.hut.fi/Software/smodels
27. Wang S., Zhang Y.(2005) A Formalization of Distributed Authorization with Delegation. In *Proceedings of the 10th Australian Conference on Information Security and Privacy*, pp 303-315, LNCS 3574.
28. Woo T. Y. C., Lam S. S., Authorization in Distributed Systems: A New Approach. *Journal of Computer Security*, 2:2/3, pp. 107C136, 1993.
29. Zhang Y.(2003) Two results for prioritized logic programming. *Theory and Practice of Logic Programming*, Vol 3(2): 223-242.

Appendix A

The program for the scenario 1 in section 4:

```
time(1..6).
subjects(alice;bob;carol;david).
gsubs(l).

#domain subjects(X;Y;Z).
#domain gsubs(L).
#domain time(T;T1;T2).

% Beginning of translation
assert(hrM,isAManager(alice)).
assert(hrM,isAnAuditor(bob)).
```

```
assert(hrM,isAnAuditor(carol)).
assert(hrM,isATech(david)).

% For auth transformation.
auth(local, l, right(pp,recovery,key),1).
match(l,right(pp,recovery,key)):-
     auth(local,l,right(+,recovery,key),1),
     1{req(X,right(pp,recovery,key)):
            assert(hrM,isAManager(X))}1,
     1{req(Y,right(pp,recovery,key)):
            assert(hrM,isAnAuditor(Y))}1,
     1{req(Z,right(pp,recovery,key)):
            assert(hrM,isATech(Z))}1.

% Request transformation
req(alice,right(pp,recovery,key)).
req(bob,right(pp,recovery,key)).
req(david,right(pp,recovery,key)).

% Authorization rule.
exist_pos(L,right(pp,recovery,key)):-
      auth(local,L,right(pp,recovery,key),T).
exist_neg(L,right(mm,recovery,key)):-
      auth(local,L,right(mm,recovery,key),T).
ggrant(L,right(pp,recovery,key)):-
      auth(local,L,right(pp,recovery,key),T),
      match(L,right(pp,recovery,key)),
      not exist_neg(L,right(mm,recovery,key)).

ggrant(L,right(mm, recovery,key)):-
      not ggrant(L,right(pp,recovery,key).
```

Appendix B

The program for scenario 2 in section 4.

```
lenth(0..5). sign(pp;mm).
obj(http;smtp;ftp;mysql;services).
sub(alice;bob;local;so;hrM).

#domain lenth(T;T1;T2).
#domain lenth(Dep;Dep1).
#domain sign(Sn).
#domain obj(GO;O).
#domain sub(X;Y;Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
below(http, services).
below(mysql, services).
below(smtp,services).
below(ftp, services).

assert(hrM, isStaff(alice)).
assert(hrM, isStaff(bob)).
assert(hrM, onHoliday(alice)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
delegate(local, so,
     right(both,access,services),3,1).
%% add implied rules.
delegate(local, so, right(both,access,O),3,1):-
    delegate(local,so,
        right(both,access,services),3,1),
    below(O,services).

auth(local,X,right(Sn,access,services),T+1):-
    delegate(local,so,right(both,access,O),3,1),
    auth(so,X,right(Sn,access,O),T).

delegate(local,X,
        right(both,access,O),min(3-T,Dep),1+T):-
    delegate(local,so,right(both,access,O),3,1),
    delegate(so,X,right(both,access,O),Dep,T),
    T < 3.

delegate(so,so,right(pp,access,services),Dep,1):-
    delegate(local,so,
        right(both,access,services),3,1),
    Dep <= 3.

delegate(local,so,
        right(both,access,services),Dep,1):-
    delegate(local,so,
        right(both,access,services),3,1),
    Dep < 3.

auth(so,X,right(pp,access,Q),1):-
    assert(hrM,isStaff(X)),
    below(Q,services), neq(Q,mysql).

auth(so,X,right(pp,access,mysql),1):-
    assert(hrM,isStaff(X)),
    not assert(hrM,onHoliday(X)).

% authorization rules.
exist_pos(X,right(pp,access,O)):-
    auth(local,X,right(pp,access,O),T).

exist_neg(X,right(mm,access,O)):-
    auth(local,X,right(mm,access,O),T).

grant(X,right(pp,access,O)):-
    auth(local,X,right(pp,access,O),T),
    not exist_neg(X,right(mm,access,O)).

grant(X,right(mm,access,O)):-
    not exist_pos(X,right(pp,access,O)).

%conflict rules.
pos_far(X,right(pp,access,O),T1):-
    auth(local, X, right(pp,access,O),T1),
    auth(local, X, right(mm,access,O),T2),
    T1>T2.

neg_far(X,right(mm,access,O),T1):-
    auth(local, X, right(mm,access,O),T1),
    auth(local, X, right(pp,access,O),T2),
    T1>T2.

grant(X,right(pp,access,O)):-
    auth(local, X, right(mm,access,O),T1),
    neg_far(X,right(mm,access,O),T1),
    auth(local, X, right(pp,access,O),T2),
    not pos_far(X, right(pp,access,O),T2).

grant(X,right(mm,access,O)):-
    auth(local,X,right(pp,access,O),T1),
    auth(local, X, right(mm,access,O),T2),
    not neg_far(X, right(mm,access,O),T2).
```

Shujing Wang received her Bachelor of Science in Physics from Wuhan University, China, in 1995 and her Master of Engineering from the Chinese Academy of Sciences, China, in 1998. Since then, she had worked as a software engineer in Institute of Software, Chinese Academy of Sciences for four and a half years. Currently she is pursuing her PhD in Computer Science at the University of Western Sydney, Australia, in the area of logic programming based formal representations for authorization and security protocols.

Yan Zhang is an associate professor in the School of Computing and Mathematics, University of Western Sydney, having received his PhD degree in computer science from Sydney University in 1994. He has research interests in knowledge representation, logic programming, model checking, descriptive complexity theory, and information security. Yan Zhang has published an impressive number of significant research papers in top international conferences and journals in his areas.